03 | RDD：为什么你必须要理解弹性分布式数据集？

2021-03-19 吴磊

Spark性能调优实战 进入课程 >

讲述：吴磊
时长 15:22 大小 14.09M

你好，我是吴磊。

从今天开始，我们进入原理篇的学习。我会以性能调优为导向，给你详细讲讲 Spark 中的核心概念 RDD 和 DAG，以及重要组件调度系统、存储系统和内存管理。这节课，咱们先来说说 RDD。

RDD 可以说是 Spark 中最基础的概念了，使用 Spark 的开发者想必对 RDD 都不陌生，甚至提起 RDD，你的耳朵可能都已经听出茧子了。不过，随着 Spark 开发 API 的演进和发展，现在上手开发基本都是 DataFrame 或 Dataset API。所以很多初学者会认为，“RDD API 基本都没人用了，我也没必要弄明白 RDD 到底是什么。”

真的是这样的吗？当然不是。

RDD 为何如此重要

首先，RDD 作为 Spark 对于分布式数据模型的抽象，是构建 Spark 分布式内存计算引擎的基石。很多 Spark 核心概念与核心组件，如 DAG 和调度系统都衍生自 RDD。**因此，深入理解 RDD 有利于你更全面、系统地学习 Spark 的工作原理。**

其次，尽管 RDD API 使用频率越来越低，绝大多数人也都已经习惯于 DataFrame 和 Dataset API，但是，无论采用哪种 API 或是哪种开发语言，你的应用在 Spark 内部最终都会转化为 RDD 之上的分布式计算。换句话说，**如果你想要在运行时判断应用的性能瓶颈，前提是你要对 RDD 足够了解**。还记得吗？定位性能瓶颈是 Spark 性能调优的第一步。

不仅如此，对于 RDD 不求甚解还有可能带来潜在的性能隐患，接下来，我们就从一个反例入手，一起来分析一下。

还记得，我们在第 1 讲中讲过的数据过滤与聚合的反例吗？通过这个例子我们明白了性能调优的必要性。那这个例子和 RDD 有什么关系呢？

别着急，我们先来回顾一下这个案例中的代码实现，去挖掘开发者采用这种实现方式的深层原因。

复制代码

```
  //实现方案1 —— 反例
def createInstance(factDF: DataFrame, startDate: String, endDate: String): Dat
val instanceDF = factDF
    .filter(col("eventDate") > lit(startDate) && col("eventDate") <= lit(endDa
    .groupBy("dim1", "dim2", "dim3", "event_date")
    .agg("sum(value) as sum_value")
instanceDF
}

pairDF.collect.foreach{
case (startDate: String, endDate: String) =>
    val instance = createInstance(factDF, startDate, endDate)
    val outPath = s"${rootPath}/endDate=${endDate}/startDate=${startDate}"
    instance.write.parquet(outPath)
}

```

在这段代码中，createInstance 的主要逻辑是按照时间条件对 factDF 进行过滤，返回汇总的业务统计量，然后 pairDF 循环遍历每一对开始时间和结束时间，循环调用 createInstance 获取汇总结果并落盘。我们在第 1 课中分析过，这份代码的主要问题在于囊括上千万行数据的 factDF 被反复扫描了几百次，而且是全量扫描，从而拖垮了端到端的执行性能。

那么，我们不禁要问：开发者究竟为什么会想到用这种低效的方式去实现业务逻辑呢？或者说，是什么内驱因素让开发者自然而然地采用这种实现方式呢？

让我们跳出 Spark、跳出这个专栏，把自己置身于一间教室内：黑板前，老师正在讲解《XX 语言编程》，旁边是你的同学，他边听老师讲课，边翻看着桌上的课本。这个场景熟不熟悉？亲不亲切？回想一下，老师讲的、书本上教的和我们示例中的代码，是不是极其类似？

没错！我们的大脑，已经习惯了 for 循环，习惯了用函数处理变量、封装计算逻辑，习惯了面向过程的编程模式。在分布式计算出现以前，我们都是这么开发的，老师也是这么讲的，书本上也是这么教的，没毛病。

因此我认为，开发者之所以会选择上面的实现方式，根本原因在于他把 factDF 当成了一个普通变量，一个与 createInstance 函数中 startDate、endDate 同等地位的形参，他并没有意识到，factDF 实际上是一个庞大的、横跨所有计算节点的分布式数据集合，更没有意识到，在分布式运行环境中，外面的 for 循环会导致这个庞大的数据集被反复地全量扫描。

这种对于分布式计算认知方面的缺失，究其缘由，还是我们对 Spark 核心概念 RDD 的理解不够透彻。所以你看，深入理解 RDD 还是很有必要的，**对于 RDD 一知半解，极有可能在应用开发的过程中，不知不觉地留下潜在的性能隐患。**

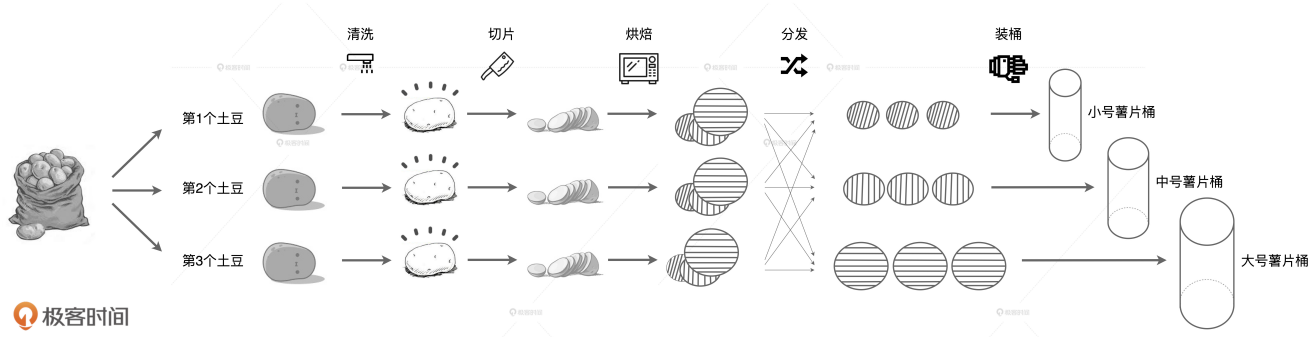
深入理解 RDD

既然 RDD 如此重要，它究竟是什么呢？2010 年，在一个夜黑风高的夜晚，Matei 等人发表了一篇名为《Spark: Cluster Computing with Working Sets》的论文并首次提出了 RDD 的概念。RDD，全称 Resilient Distributed Datasets，翻译过来就是弹性分布式数据集。本质上，它是对于数据模型的抽象，用于囊括所有内存中和磁盘中的分布式数据实体。

如果就这么从理论出发、照本宣科地讲下去，未免过于枯燥、乏味、没意思！不如，我先来给你讲个故事。

从薯片的加工流程看 RDD

在很久很久以前，有个生产桶装薯片的工坊，工坊的规模较小，工艺也比较原始。为了充分利用每一颗土豆、降低生产成本，工坊使用 3 条流水线来同时生产 3 种不同尺寸的桶装薯片。3 条流水线可以同时加工 3 颗土豆，每条流水线的作业流程都是一样的，分别是清洗、切片、烘焙、分发和装桶。其中，分发环节用于区分小、中、大号 3 种薯片，3 种不同尺寸的薯片分别被发往第 1、2、3 条流水线。具体流程如下图所示。



RDD的生活化类比

看得出来，这家工坊制作工艺虽然简单，倒也蛮有章法。从头至尾，除了分发环节，3 条流水线没有任何交集。在分发环节之前，每条流水线都是专心致志、各顾各地开展工作：把土豆食材加载到流水线上，再进行清洗、切片、烘焙；在分发环节之后，3 条流水线也是各自装桶，互不干涉、互不影响。流水线的作业方式提供了较强的容错能力，如果某个加工环节出错，工人们只需要往出错的流水线上重新加载一颗新的土豆，整个流水线就能够恢复生产。

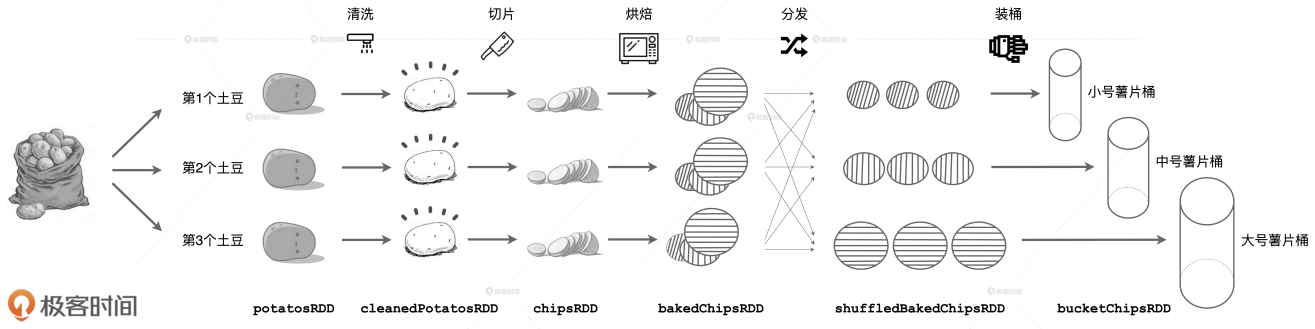
好了，故事讲完了。如果我们把每一条流水线看作是分布式运行环境的计算节点，用薯片生产的流程去类比 Spark 分布式计算，会有哪些有趣的发现呢？

仔细观察，我们发现：**刚从地里挖出来的土豆食材、清洗过后的干净土豆、生薯片、烤熟的薯片，流水线上这些食材的不同形态，就像是 Spark 中 RDD 对于不同数据集合的抽象。**

沿着流水线的纵深方向，也就是图中从左向右的方向，每一种食材形态都是在前一种食材之上用相应的加工方法进行处理得到的。**每种食材形态都依赖于前一种食材，这就像是**

RDD 中 dependencies 属性记录的依赖关系，而不同环节的加工方法，对应的刚好就是 RDD 的 compute 属性。

横看成岭侧成峰，再让我们从横向的角度来重新审视上面的土豆加工流程，也就是图中从上至下的方向，让我们把目光集中在流水线开端那 3 颗带泥的土豆上。这 3 颗土豆才从地里挖出来，是原始的食材形态，正等待清洗。如图所示，我们把这种食材形态标记为 potatosRDD，那么，**这里的每一颗土豆就是 RDD 中的数据分片，3 颗土豆一起对应的就是 RDD 的 partitions 属性。**



带泥土豆经过清洗、切片和烘焙之后，按照大小个儿被分发到下游的 3 条流水线上，这 3 条流水线上承载的 RDD 记为 shuffledBakedChipsRDD。很明显，这个 RDD 对于 partitions 的划分是有讲究的，根据尺寸的不同，即食薯片会被划分到不同的数据分片中。**像这种数据分片划分规则，对应的就是 RDD 中的 partitioner 属性。** 在分布式运行环境中，partitioner 属性定义了 RDD 所封装的分布式数据集如何划分成数据分片。

总的来说，我们发现，薯片生产的流程和 Spark 分布式计算是一一对应的，一共可以总结为 6 点：

土豆工坊的每条流水线就像是分布式环境中的计算节点；

不同的食材形态，如带泥的土豆、土豆切片、烘烤的土豆片等等，对应的就是 RDD；

每一种食材形态都会依赖上一种形态，如烤熟的土豆片依赖上一个步骤的生土豆切片。这种依赖关系对应的就是 RDD 中的 dependencies 属性；

不同环节的加工方法对应 RDD 的 compute 属性；

同一种食材形态在不同流水线上的具体实物，就是 RDD 的 partitions 属性；

食材按照什么规则被分配到哪条流水线，对应的就是 RDD 的 partitioner 属性。

不知道土豆工坊的类比，有没有帮你逐渐勾勒出 RDD 的本来面貌呢？话付前言，接下来，咱们来一本正经地聊聊 RDD。

RDD 的核心特征和属性

通过刚才的例子，我们知道 RDD 具有 4 大属性，**分别是 partitions、partitioner、dependencies 和 compute 属性。正因为有了这 4 大属性的存在，让 RDD 具有分布式和容错性这两大最突出的特性。**要想深入理解 RDD，我们不妨从它的核心特性和属性入手。

首先，我们来看 partitions、partitioner 属性。

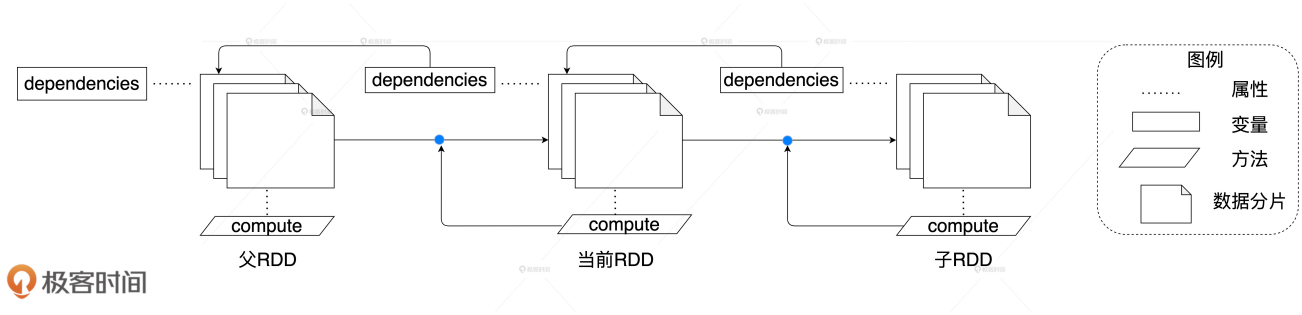
在分布式运行环境中，RDD 封装的数据在物理上散落在不同计算节点的内存或是磁盘中，这些散落的数据被称“数据分片”，RDD 的分区规则决定了哪些数据分片应该散落到哪些节点中去。RDD 的 partitions 属性对应着 RDD 分布式数据实体中所有的数据分片，而 partitioner 属性则定义了划分数据分片的分区规则，如按哈希取模或是按区间划分等。

不难发现，partitions 和 partitioner 属性刻画的是 RDD 在跨节点方向上的横向扩展，所以我们把它们叫做 RDD 的“横向属性”。

然后，我们再来说说 dependencies 和 compute 属性。

在 Spark 中，任何一个 RDD 都不是凭空产生的，每个 RDD 都是基于某种计算逻辑从某个“数据源”转换而来。RDD 的 dependencies 属性记录了生成 RDD 所需的“数据源”，术语叫做父依赖（或父 RDD），compute 方法则封装了从父 RDD 到当前 RDD 转换的计算逻辑。

基于数据源和转换逻辑，无论 RDD 有什么差池（如节点宕机造成部分数据分片丢失），在 dependencies 属性记录的父 RDD 之上，都可以通过执行 compute 封装的计算逻辑再次得到当前的 RDD，如下图所示。



基于dependencies和compute属性得到当前RDD

由 dependencies 和 compute 属性提供的容错能力，为 Spark 分布式内存计算的稳定性打下了坚实的基础，这也正是 RDD 命名中 Resilient 的由来。接着观察上图，我们不难发现，不同的 RDD 通过 dependencies 和 compute 属性链接在一起，逐渐向纵深延展，构建了一张越来越深的有向无环图，也就是我们常说的 DAG。

由此可见，dependencies 属性和 compute 属性负责 RDD 在纵深方向上的延展，因此我们不妨把这两个属性称为“纵向属性”。

总的来说，**RDD 的 4 大属性又可以划分为两类，横向属性和纵向属性。其中，横向属性锚定数据分片实体，并规定了数据分片在分布式集群中如何分布；纵向属性用于在纵深方向构建 DAG，通过提供重构 RDD 的容错能力保障内存计算的稳定性。**

同时，为了帮助你记忆，我把这 4 大核心属性的基本概念和分类总结在了如下的表格中，你可以看一看。

属性名	成员类型	属性含义	RDD特性	刻画方向
partitions	变量	RDD的所有数据分片实体	分布式	横向
partitioner	方法	划分数据分片的规则		
dependencies	变量	生成该RDD所依赖的父RDD	容错性	纵向
compute	方法	生成该RDD的计算接口		

极客时间

除此之外，我还想再多说两句。在这节课开头的反例中，我们分析了开发者采用 foreach 语句循环遍历分布式数据集的深层次原因。**这种不假思索地直入面向过程编程、忽略或无视分布式数据实体的编程模式，我将其称为单机思维模式。**

在学习了 RDD 横向的 partitions 属性和纵向的 dependencies 属性之后，如果你能把它们牢记于心，那么在频繁调用或引用这个 RDD 之前，你自然会想到它所囊括的数据集合，很有可能在全节点范围内被反复扫描、反复计算。这种下意识的反思会驱使你尝试探索其他更优的实现方式，从而跳出单机思维模式。因此，**深入理解 RDD，也有利于你跳出单机思维模式，避免在应用代码中留下性能隐患。**

小结

今天，我带你学习了 RDD 的重要性，以及它的 2 大核心特性和 4 大属性。

首先，深入理解 RDD 对开发者来说有百利而无一害，原因有如下 3 点：

Spark 很多核心概念都衍生自 RDD，弄懂 RDD，有利于你全面地学习 Spark；

牢记 RDD 的关键特性和核心属性，有利于你在运行时更好地定位性能瓶颈，而瓶颈定位，恰恰是性能调优的前提；

深入理解 RDD 有利于你跳出单机思维模式，避免在应用代码中留下性能隐患。

关于 RDD 的特性与核心属性，只要你把如下 2 点牢记于心，我相信在不知不觉中你自然会绕过很多性能上的坑：

横向属性 partitions 和 partitioner 锚定数据分片实体，并且规定了数据分片在分布式集群中如何分布；

纵向属性 dependencies 和 compute 用于在纵深方向构建 DAG，通过提供重构 RDD 的容错能力保障内存计算的稳定性。

每日一练

1. 在日常的开发工作中，你遇到过“单机思维模式”吗？有哪些呢？
2. 除了我们今天讲的 4 大属性，RDD 还有个很重要的属性：preferredLocations。按照经验，你认为在哪些情况下，preferredLocations 很重要，会提升 I/O 效率，又在哪些

环境中不起作用呢？为什么？

期待在留言区看到你的思考，也欢迎你分享工作中遇到过的“单机思维模式”，我们下节课见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 02 | 性能调优的本质：调优的手段五花八门，该从哪里入手？

下一篇 04 | DAG与流水线：到底啥叫“内存计算”？

精选留言 (12)

写留言

Shockang
2021-03-19

RDD的5大特性总结起来就是：3个列表，2个函数。3个列表是分区列表，依赖列表和优先位置列表；2个函数就是：计算函数和分区函数。我这样总结是为了方便记忆的，分享给大家！

作者回复: 不错的技巧～

14

Z宇锤锤
2021-03-27

RDD的四大特性：两横两纵一漂移。两纵土豆每一个生产线的流转就是父RDD到子RDD的流转，其中的流转方法就是compute算子。两横：就是土豆和土豆片之间的工具，切出来的土豆片就是partitions,怎么切土豆片就是partitoner。漂移就是prefered location,选择在哪个RDD上取数据。

展开

作者回复: 66666666666666666 x 10的6次幂，全场你最6～

1 5

joan
2021-03-23

preferredLocations不起作用的场景有一种是存算分离的场景，计算节点在k8s集群里，远程读取分布式文件系统，例如hdfs或者ceph等，可能接入万兆SDN网络能够有一些收益

作者回复: 正解～

3 3

Z宇锤锤
2021-03-27

关于RDD的单机思维模式
RDD中的集合排序问题
RDD.collect.sort

展开

作者回复: 可以的，相当精辟～

2

走刀口

2021-03-28

老师，请教个问题，关于宽窄依赖划分。网上很多文章是根据父子rdd分区数量划分的，但我这样不对。比如cartsian算子，它父rdd一个分区对应子rdd多个分区，可它是窄依赖。
宽窄依赖划分依据到底是啥?

展开

作者回复: 是的，不能单纯根据分区数完成判断。

看算子，分类讨论。一类算子会直接记录宽窄依赖关系，另一类算子通过判断partitioner与父依赖是否一致来决定。

你可以在源码的RDD目录下查一查，哪些算子属于第一类，哪些属于第二类～

1

林建林

2021-03-31

preferredLocations：可以理解为最佳位置，移动计算不移动数据，数据在哪就在哪计算，减少IO

作者回复: 对，是这个道理，不过要考虑，有些情况，可以数据不动代码动，有些情况，就必须要动数据了。这也是这个问题想要大家去深入思考的，要分类讨论。

Insomnia

2021-03-26

老师, 我这么理解不知道对不对?
当下比较流行的计算框架 Spark/Flink, 如果使用 Yarn 来调度计算任务, 对于 Yarn 而言, 本地的 Spark/Flink 都可以看作是 Yarn 的客户端, 而对于用户而言, 编写的 job 都可以看做是本地的 Spark/Flink 的客户端, 他们共同的本质都是使用本地 Spark/Flink, 将用户的 job 解析并优化成物理执行计划, 然后提交到 Yarn 上部署执行

展开

作者回复: 没错，理解的非常到位！本质上就是这么个过程。

当然yarn也有yarn client和yarn cluster模式哈，不过本质上区别不大，无非是app master / driver放哪的问题，都适用你说的那个过程。

Geek_d794f8
2021-03-25

老师，如果计算和存储分离，比如计算集群A，存储集群B，且spark计算基于yarn调度，那么是不是，yarn集群必须部署在集群A上?

展开

作者回复: 反过来说可能更合适，你有个yarn集群，那么就可以拿来给spark应用调度资源，甚至不需要额外部署什么“spark”集群。一般说spark集群，通常指的是工作在standalone调度模式下的集群。当然，如果你本身有一个standalone的spark集群，想换用yarn做资源调度，那么就像你说的，yarn要和spark部署在同一个集群。

弦断觅知音
2021-03-24

preferredLocations不起作用情况之一，比如有单独机器作为集群的客户端，然后提交的任务在集群的客户端上运行，这种情况算不算?

展开

作者回复: 这种不算哈，可以沿着计算存储分离的思路再想想～

zxk
2021-03-20

preferredLocations，作业尽可能在在数据所在节点上运行，或者同个机架下等情况，都可以提升 I/O 效率。
当数据不管落在哪个节点，都需要从远端拉取数据时，这时候作用可能就不是那么大。

作者回复: 正解，能举例说明吗?

2

L3nvy
2021-03-19

1. “单机模式”，不知道算不算。不知道Spark可以开多线程触发action算子，同时提交多个不相关的作业，Spark UI同时运行的job拉满 :)。之前代码都是按面向过程的思维一步一步写的。
2. 处理数据源的时候，jdbc，读取本地文件不起作用

展开

作者回复: 1. 算的，非常好的单机思维案例，凡是上来就面向过程，把分布式数据集rdd、dataframe、dataset当成普通单机变量的，都是单机思维的一种表现。不停的调action，我怀疑是想迫切地看到计算结果，忽略了action会触发dag回溯。

2.没错！通过jdbc远程访问db通常没办法保证本地性，沿着计算和存储分离的思路，再想一想，还有其他案例吗?

4

老A
2021-03-19

preferredLocations 不起作用，应该在数据源是单节点情况，比如socket数据源，还有就是读取hdfs的数据块比较小不足一个split分片128M的情况，还有就是大的压缩数据块，比如snappy压缩，这种压缩不支持切分的情况。

展开

作者回复: 这块确实有点难，因为我们还没讲到调度系统，也还没细讲locality。再想一想，沿着计算和存储物理分离的思路哈～